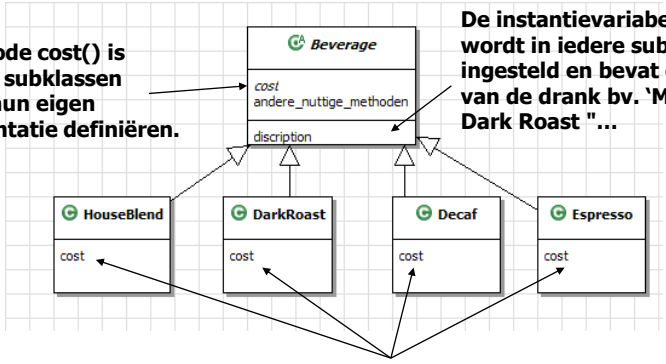
Decorator Pattern – structuur van objecten

Voorbeeld "Starbuzz Coffee"

Toen ze met hun business begonnen, ontwierpen ze hun klassen als volgt...

Beverage is een abstracte klasse waar alle dranken die verkrijgbaar zijn in de taverne als subklassen van afstammen.

De methode cost() is abstract; subklassen moeten hun eigen implementatie definiëren.

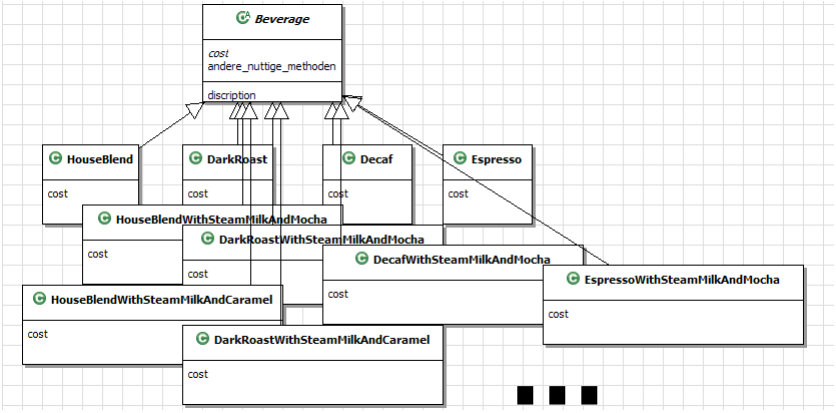


De instantievariabele description wordt in iedere subklasse ingesteld en bevat een beschrijving van de drank bv. 'Most Excellent Dark Roast "...

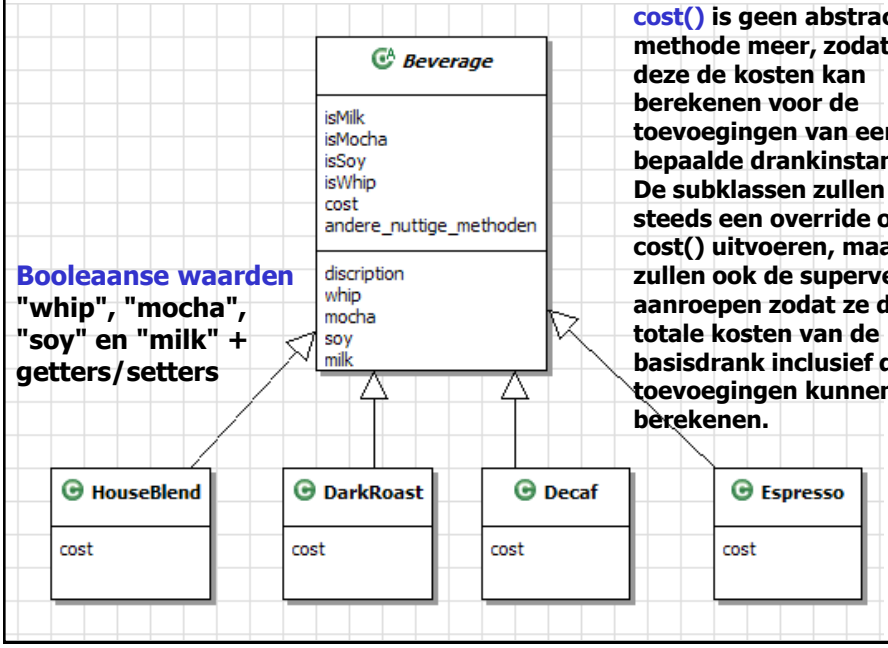
Iedere subklasse implementeert cost() om de kosten van de drank te retourneren.

- **Behalve koffie, kun je ook om diverse toevoegingen vragen, zoals gestoomde melk, soja, mokka en alles bedekt met een laag geklopte melk. Starbuzz rekent voor alles iets extra, deze zaken moeten dus werkelijk in hun ordersysteem worden opgenomen.**

Hier volgt hun eerste poging …



Waarom hebben we al die subklassen nodig? Kunnen we niet gewoon instantievariabelen en overerving gebruiken?



Booleaanse waarden "whip", "mocha", "soy" en "milk" + getters/setters

cost() is geen abstracte methode meer, zodat deze de kosten kan berekenen voor de toevoegingen van een bepaalde drankinstantie. De subklassen zullen nog steeds een override op cost() uitvoeren, maar zullen ook de superversie aanroepen zodat ze de totale kosten van de basisdrank inclusief de toevoegingen kunnen berekenen.

Correcte oplossing?

Welke eisen of andere factoren kunnen veranderen en zijn van invloed op dit ontwerp?

Prijsverandering van de toevoegingen dwingen ons de bestaande code aan te passen.

Nieuwe toevoegingen dwingen ons nieuwe methoden op te nemen en de methode cost() in de superklasse aan te passen.

We kunnen nieuwe dranken krijgen. Bv. ice tea -> zal methode isMilk() erven.

Wat als een klant een dubbele mokka wil?

Het open-geslotenprincipe

Klassen moeten open zijn voor uitbreiding, doch gesloten voor verandering.

Ons doel is dat klassen eenvoudig uitgebreid kunnen worden om nieuw gedrag te incorporeren zonder de bestaande code te wijzigen. Hoe we dat gaan bereiken? **Door ontwerpen te maken die weerstand bieden tegen verandering en flexibel genoeg zijn om nieuwe functionaliteiten op te nemen om aan veranderende eisen tegenmoet te komen**.

6

Het open-geslotenprincipe

Ook al klinkt het tegenstrijdig, er bestaan technieken om code uit te breiden zonder die rechtstreeks te moeten wijzigen.

Wees voorzichtig bij je keuze van de codegebieden die uitgebreid moeten worden: het OVERAL toepassen van het open-geslotenprincipe kan onnodig kostbaar zijn en kan leiden tot complexe, moeilijk te begrijpen code.

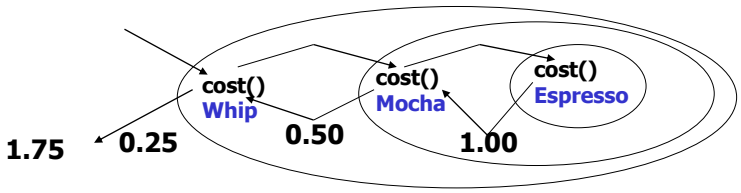
Maak kennis met het Decorator Pattern

Bv.
1) Het Espresso-object nemen;

2) dit met een Mocha-object decoreren;

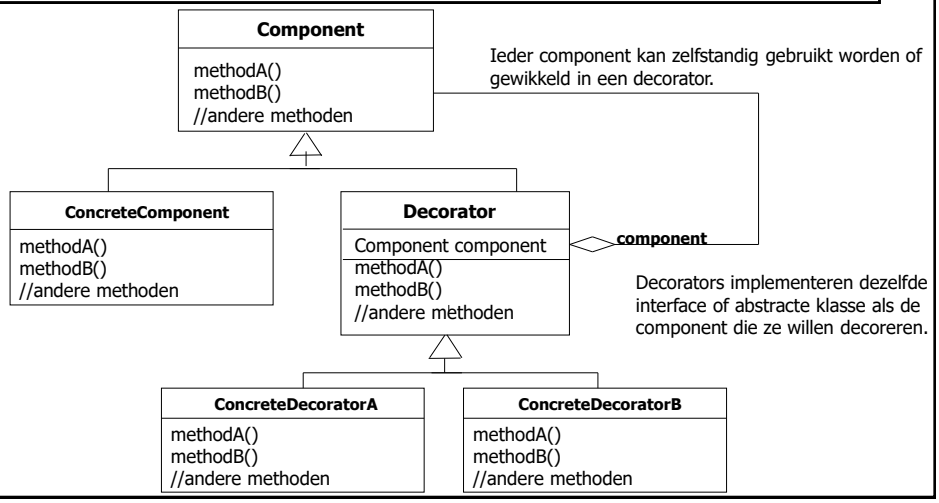
3) dit met een Whip-object decoreren;

4) de methode cost() aanroepen en er op vertrouwen dat door het delegeren de kosten voor de toevoegingen worden bijgeteld.

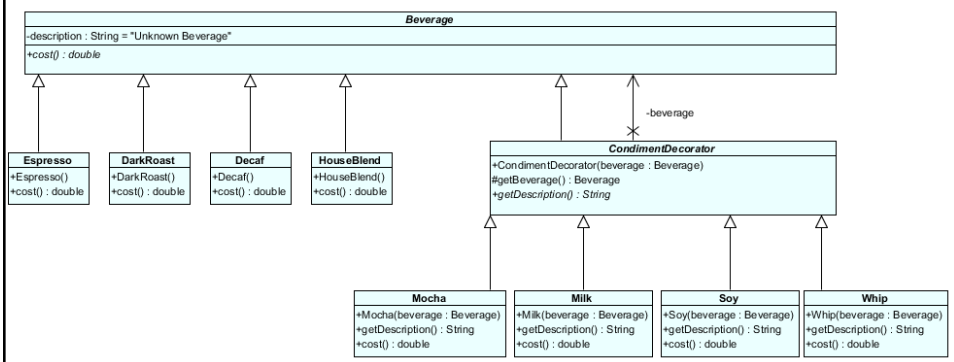


Decorator – structuur van objecten

Het Decorator Pattern kent dynamisch additionele verantwoordelijkheden toe aan een object. Decorators bieden een flexibel alternatief voor het gebruik van subklassen om functionaliteiten uit te breiden.



Onze Beverages decoreren



De Startbuzz-code schrijven

```
public abstract class Beverage
{
    private String description = "Unknown Beverage";

    public String getDescription() { return description;}
    protected void setDescription(String description) {
        this.description = description;
    }
    public abstract double cost();
}
```

We moeten uitwisselbaar blijven met Beverage, dus erven we van de klasse Beverage.

```
public abstract class CondimentDecorator extends Beverage
{   private final Beverage beverage;
    public CondimentDecorator(Beverage beverage) {
        this.beverage = beverage;
    }
    protected Beverage getBeverage() { return beverage; }
    public abstract String getDescription();
}
```

De Startbuzz-code schrijven

OOAD

```
public class Espresso extends Beverage {

    public Espresso() {
        setDescription("Espresso");
    }

    public double cost() {
        return 1.99;
    }

}
```

Zorg voor de juiste beschrijving

De kosten van een Espresso berekenen.

Analoog voor de klassen "HouseBlend", "DarkRoast" en "Decaf"

```
/*Milk is een decorator, dus erven we van CondimentDecorator.
(CondimentDecorator is een uitbreiding van Beverage)*/
public class Milk extends CondimentDecorator {

    //De beverage die we omwikkelen, dragen we over aan de
        //constructor van de decorator.
    public Milk(Beverage beverage) {
        super(beverage);
    }

    public String getDescription() {
        return getBeverage().getDescription() + ", Milk";
    }

    public double cost() {
        return .10 + getBeverage().cost();
    }
}
```

Koffie serveren


```
//imports
public class StarbuzzCoffee{

    public StarbuzzCoffee() {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + " $" + beverage.cost());

        Beverage beverage2 =
                new Whip( new Mocha( new Mocha( new DarkRoast())));
        System.out.println(beverage2.getDescription()
                + " $" + beverage2.cost());
    }
}
```

```
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
```

Bestaande Decorators: Java I/O

Het grote aantal klassen in de package java.io is overweldigend. Maar nu je het Decorator Pattern kent, komen deze I/O-klassen vermoedelijker duidelijker over.

Bv. ObjectInputStream ois =
new ObjectInputStream(
new FileInputStream(naamBestand))